Université de Bordeaux

Mémoire

Fixation d'un drone par électro-aimant

Master 2 ASPIC

Tom Déporte, Jessy Nolibé

Client : Serge Chaumette

Chargé de TD : Desbarats Pascal

28 mars 2022

Table des matières

Chapitre 1

Introduction au projet

Ce mémoire a été réalisé dans le cadre de l'UE : **Projet de Fin d'Etude** dans laquelle il nous a été confié une mission consistant à développer le système complet d'un drone qui a la capacité de s'accrocher / décrocher à la demande via un électro-aimant.

Ce projet sera livré à notre client **Serge Chaumette**.
Il est également suivi par **Desbarats Pascal** qui, dans le cadre de l'UE **MOCPI**, nous guide pour le mener à bien en suivant une méthode de travail agile.

1.1 Rappel de la problématique du projet et Motivations

Les drones sont des outils utilisés dans de multiples domaines pour réaliser des missions de tous types. Les innovations sur ces appareils sont très nombreuses, ce qui amène de nouvelles problématiques dont la suivante : Comment faciliter le déploiement d'un drone tout en optimisant l'espace dans lequel il opère ?
Selon les cas d'usages, il est parfois nécessaire que les drones puissent être déployés dans un laps de temps très court.
Or pour pouvoir le garantir, les appareils doivent être constamment dans une zone permettant le décollage. Une zone particulière doit donc être allouée au-dit décollage.

Le déploiement de drones lors de missions, notamment en intérieur, pour des services autonomes et réguliers (surveillance, livraison...) impliquent le besoin d'une surface dédiée au décollage et à l'atterrissage, qui peut être contraignante et encombrante.

Afin d'y remédier une solution intéressante serait d'utiliser un espace qui n'est pas encore sollicité, tel que le plafond. Une installation permettant de fixer son appareil à la demande au plafond permettrait de retirer le besoin de préparer l'utilisation et donc augmenterait la vitesse de déploiement du drone.

Chapitre 2

Users Stories et Besoins

2.1 User Stories

Au terme du projet, trois fonctionnalités doivent être présentes pour les utilisateurs.

* En tant qu'utilisateur je souhaite pouvoir ordonner au drone d'aller se fixer sur la plaque métallique.
* En tant qu'utilisateur je souhaite pouvoir ordonner au drone de se détacher de son support.
* En tant qu'utilisateur je souhaite que le drone soit capable de détecter son support afin de faciliter sa fixation.

2.2 Besoins fonctionnels

Détection de la plaque

* Le drone aurait la fonctionnalité de pouvoir repérer sa plaque de fixation pour permettre d'automatiser le retour sur son support à la fin de la mission qui lui a été confié. Cette fonction de repérage se ferait via un QR Code placé sur la plaque et une caméra sur le haut du drone permettant de faire du traitement d'image pour détecter et calculer la position de la plaque.

Fixation du drone

* **Accrochage du drone :**
  Le premier objectif de l'appareil est d'avoir la possibilité de se fixer sur la plaque métallique pour permettre de résoudre la problématique principale de ce projet qui est d'offrir un espace dédié au déploiement d'un drone d'intérieur.
  Pour se faire, le châssis du drone sera équipé d'un électro-aimant qui se déclenchera lors du contact avec la plaque métallique. Une fois l'électro-aimant actionné le drone pourra s'arrêter et sera bien fixé au plafond.

* **Décrochage du drone :**
  De manière complémentaire à l'accroche du drone sur le support métallique, il faut que l'appareil puisse se décrocher lorsque son utilisation est requise.
  Pour réaliser ce décrochage, le drone va devoir en premier lieu se lancer avec une puissance suffisante pour lui permettre de pouvoir rester en vol stationnaire et après ceci l'électro-aimant pourra être désactivé pour laisser le drone mener à bien son objectif.

Contrôle du drone

* Pour satisfaire tous les besoins, avoir un contrôle sur le drone est nécessaire pour pouvoir démarrer ou arrêter une mission. Ce contrôle sur le drone sera possible de deux manière différentes, d'une façon manuelle c'est à dire dans laquelle l'utilisateur via un ordinateur pourra demander au drone de venir se fixer au support ou à l'inverse se détacher de la plaque pour commencer à le piloter. Et d'une façon automatique le drone devrait pouvoir retourner se fixer de lui-même lorsqu'il à atteint tous ses objectifs ou se libérer seul selon les besoins, par exemple si sa fonction est la surveillance il se libérera à intervalles réguliers pour effectuer sa tâche.

2.3 Besoins non fonctionnels

Performance

* Pour transmettre des ordres au drone une communication sans fil est utilisé entre le drone et l'appareil de l'utilisateur, il est donc important d'avoir une transmission rapide pour offrir une utilisation fluide et intéressante.

2.4 Évolutions du cahier des besoins

Notre projet étant réalisé sous forme de simulation, des contraintes sont ajoutées par rapport à un prototype. Ces contraintes ont fait évoluer notre cahier des besoins pour s'adapter au modèle de la simulation. Les besoins fonctionnels d'accrochage et de décrochage par électro-aimant sont remplacés par un contrôle direct sur la simulation, c'est à dire mise en pause de la simulation pour contraindre le drone à se bloquer sous la plaque et reprise de la simulation pour le décrocher. Nous avons décidé d'utiliser cette solution par manque de plugins pour simuler un électro-aimant et de temps/connaissances au vu de l'ampleur du travail de création d'un plugin de zéro. Ces contraintes n'existent que dans la simulation, sur le prototype final l'implémentation et le controle

d'un électro-aimant se ferait directement sans avoir à passer par un plugin.

Chapitre 3

Architecture du logiciel

3.1 Construction et principe logiciel

Dans le cadre de notre projet, l'objectif final était de s'approcher le plus possible d'un prototype fonctionnel. Pour simplifier la programmation et les tests, tout en restant proche d'un prototype la meilleure solution était d'utiliser un logiciel de simulation dont le code est presque portable sur l'appareil tel quel. Pour notre drone, en réalité nous utilisons trois modules que nous combinons pour obtenir une simulation réaliste.

* **ROS : Robot Operating System**
  Ce premier outil pour réaliser notre simulation est le plus important, il s'agit d'un système d'exploitation pour notre drone. Il offre des fonctionnalités pour pouvoir communiquer avec les robots via un système de communication asynchrone qui sont les **Topics** et un synchrone : **Service**.
  Un topic est un système de transformation de l'information basé sur le principe d'abonnement (Subscriber) et de publication (Publisher). Le principal moyen de communication que l'on a avec notre drone sont des topics, notre appareil est abonné à certains topics et nous publions dessus pour le contrôler. En conclusion ROS est l'outil nous permettant de faire le lien entre tout ce qui compose le drone et notre code, il convertit notre code en commandes comprisent et exécutés par notre appareil.

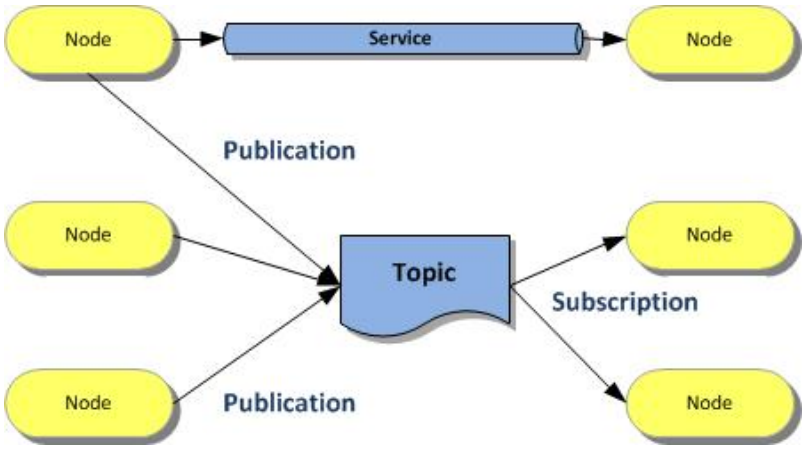


FIGURE 3.1 – Schéma Topics et Service
Source : https ://bit.ly/3wCSzpx

* **Gazebo**
  Gazebo est un logiciel open-source de simulation robotique 3D, c'est à dire qu'il simule tout un environnement avec une physique du monde réel pour pouvoir y intégrer son robot et le tester. La combinaison ROS et Gazebo permet d'obtenir une utilisation similaire à celle pouvant être obtenue sur du vrai matériel, avec un programme ROS qui peut contrôler un modèle de drone et Gazebo qui place ce même modèle dans un monde virtuelle chaque test voulant être réalisés peut être vérifié.
* **PX4**
  PX4 est un système d'auto-pilote que l'on peut intégrer dans notre drone pour automatiser et simplifier le contrôle de celui-ci. Comme on a pu le voir au dessus ROS fonctionne avec un système de topics qui font le lient entre code et contrôle. Le module PX4 va mettre à disposition des topics permettant d'envoyer des ordre au drone tel que se déplacer à telle position ou se poser... Son rôle est d'offrir une interface de contrôle qui ne nécessite pas d'avoir à programmer chaque composant individuellement.
  Si on veut que le drone se déplace à une position, on lui donne des coordonnées au lieu d'avoir à contrôler chaque rotor pour effectuer le mouvement de lacet, roulis et tangage.

3.2 Décomposition modulaire

Le dépôt de notre projet se décompose en trois répertoires.

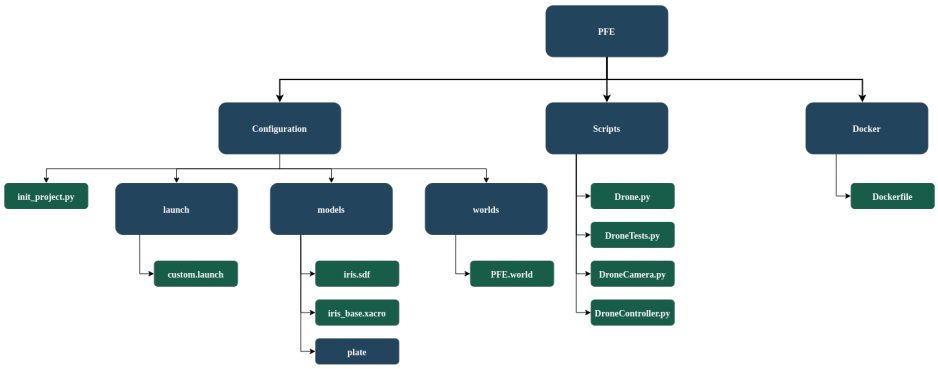


FIGURE 3.2 – Arborescence de notre projet

* **Scripts :**
  Le premier répertoire est le répertoire **scripts**/ qui contient tous les fichiers python nécessaires au contrôle de notre drone et aux tests de celui-ci.

  Drone.py est le fichier principal de ce dossier parce que c'est dans celui là que l'on retrouve la classe Drone, qui est la classe dans laquelle tout le comportement attendu du drone pour réaliser la mission est présent. Du placement de l'appareil au niveau de QR Code jusqu'a la fixation ou le décrochage de la plaque.
  DroneController.py contient le code pour configurer le drone selon ses besoins. Par exemple pour pouvoir envoyer l'appareil à une position il faut armer le drone et changer son mode pour le mode **offboard**. Ce sont des caractéristiques du fonctionnement de ROS et de l'autopilote PX4 qui sont toutes contenues dans ce fichier pour des raisons de modularité du projet.
  DroneCamera.py est un fichier dédié au capteur que l'on a ajouté qui est la caméra. Ce code python offre des méthodes permettant de démarrer le flux vidéo de la caméra, de récupérer la dernière image enregistrée et même de la sauvegarder. La fonction la plus importante est celle de récupération de l'image courante du drone parce que c'est celle-ci qui sera traité pour en déduire la position du QR Code.
  DroneTests est comme son nom l'indique le fichier de tests de tous les topics, composants, fonctions... que l'on utilise pour mener à bien la mission.

* **Configurations :**
  PX4-Autopilot est un répertoire git permettant une simulation fonctionnelle en offrant des modèles de drone et des plugins pour les contrôler plus simplement. Notre répertoire **Configuration**/ contient tous les fichiers que l'on a crées / modifiés pour notre simulation, qui seront déplacés dans le dossier de PX4 pour pouvoir lancer notre projet et profiter des fonctionnalités de l'auto-pilote.
  models/ est le dossier dans lequel se trouve tous les modèles que nous

avons conçus pour notre simulation. Notre drone est basé sur un modèle déjà existant dans PX4-Autopilot, le iris que nous avons modifié en y ajoutant une caméra et un laser via les fichiers *iris.sdf* et *iris_base.xacro*.

```
<xacro:camera_macro
  namespace="${namespace}"
  parent_link="base_link"
  camera_suffix="red_iris"
  frame_rate="30.0"
  horizontal_fov="1.3962634"
  image_width="1200"
  image_height="800"
  image_format="R8G8B8"
  min_distance="0.02"
  max_distance="1000"
  noise_mean="0.0"
  noise_stddev="0.007"
  enable_visual="1"
  >
  <box size="0.03 0.03 0.03" />
  <origin xyz="0 0 0.05" rpy="0 -1.57079 0"/>
  </xacro:camera_macro>
```

FIGURE 3.3 – Code ajouté sur le modèle iris

Nous avons également ajouté un modèle de plaque sur laquelle est imprimée un QR code créé sous Blender. Dans la simulation le QR code s'étant sur toute la plaque car nous ne sommes pas parvenu à détecter un code de plus petite taille sous Gazebo. Dans le cas où ce projet viendrait à être réalisé avec du vrai matériel, la taille du code serait une fraction de celle de la plaque.

* **worlds**/ contient le monde dans lequel se déroule notre simulation, ce dernier contient notre drone équipé d'une caméra pointée vers le haut ainsi que la plaque sur laquelle il doit venir s'accrocher. Afin de simuler une plaque métallique repérable par un QR code, nous avons créé le modèle "plate", une plaque sur laquelle est dessinée un QR code, sur Blender puis l'avons importé dans notre monde Gazebo. Le drone, à l'aide de sa caméra, peut alors repérer le centre du QR code afin d'ajuster sa position.

  launch/ contient le fichier *custom.launch* qui permet de configurer le lancement de notre monde en y ajoutant notre modèle de drone combiné avec la plaque et le QR Code ainsi que tout un tas de variables utile à Gazebo.

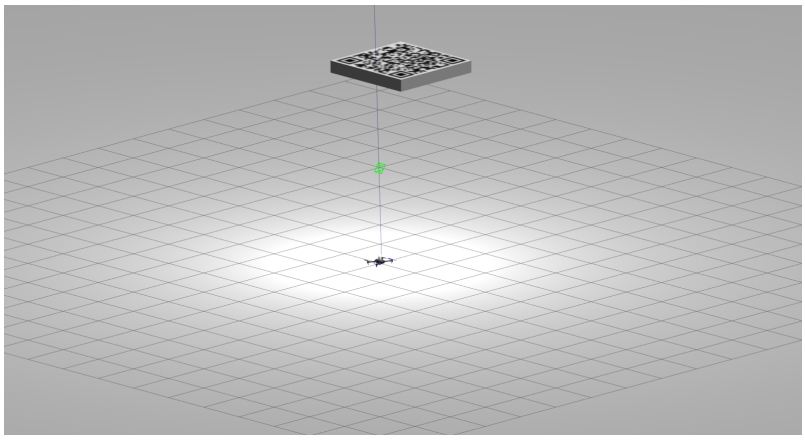


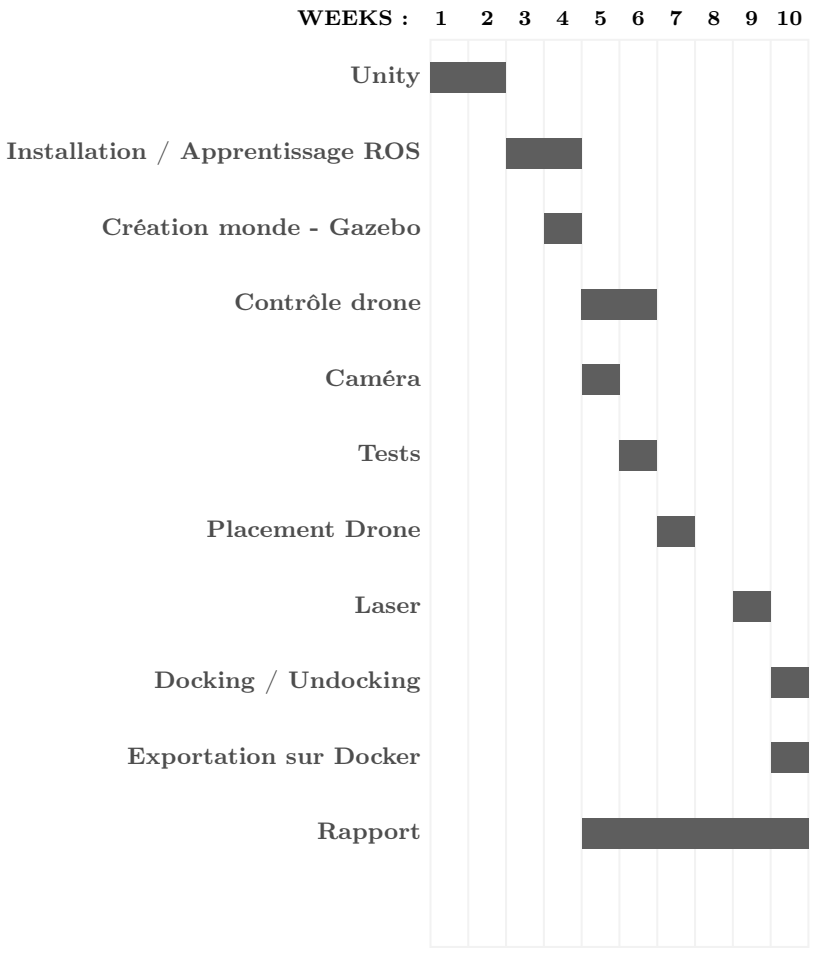
FIGURE 3.4 – Monde avec plaque et QR Code

Docker/ contient le fichier DockerFile décrivant l'image du conteneur disponible à l'adresse https://hub.docker.com/repository/docker/tdeporte/ros_sitl_gazebo, ce conteneur permet d'essayer le logiciel sans installer ROS, Gazebo, PX4 et leurs dépendances en local et nécessite uniquement Docker.

Chapitre 4

Implémentation et tests

4.1 Visualisation du projet



La répartition des tâches du gantt est la suivante :
Tom : Création monde - Gazebo, Caméra, Placement Drone, Laser, Docker
Jessy : Contrôle drone, Tests, Docking / Undocking
Tom et Jessy : Unity, Installation / Apprentissage ROS, Rapport

4.2 Explication de fonctionnement du logiciel

4.2.1 Lancement du logiciel

Le lancement et l'utilisation de la simulation est relativement simple. En premier lieu il faut ouvrir deux terminaux, dans le premier lancer la commande suivante :

```
$ roslaunch px4 custom.launch
```

et le monde gazebo contenant tous les modèles de notre simulation va se lancer.

Ce même terminal peut servir à envoyer des commandes simples au drone tel que :

```
$ commander takeoff
```

pour le faire décoller.

Dans le deuxième terminal, il faut se placer dans le dossier **scripts**/ et lancer le fichier Drone.py ou DroneTests.py via la commande :

```
$ python3 Drone.py
```

pour lancer la simulation.

```
$ python3 DroneTests.p
```

Pour lancer les tests.

Chose à savoir : Avant de lancer les tests il est préférable de lancer à côté QGroundControl car sans ça cela peut faire échouer des tests pour des raisons interne à ROS.

Après lancement de la simulation, l'utilisateur peut appuyer sur la touche **"d"** afin de démarrer le processus de recherche de la plaque.
La touche **"u"** peut être utilisée pour décrocher le drone de la plaque, le drone passe alors en mode "Atterrissage automatique".

Nous recommandons de lire le fichier README pour plus de détails.

4.3 Analyse du fonctionnement

4.3.1 Rospy et récupération de données des capteurs

Ce projet repose en grande partie sur la libraire *rospy* (disponible : http://wiki.ros.org/rospy). Cette librairie python permet de rapidement accéder aux valeurs partagées sur divers rostopics.

4.3.2 Récupération de la capture de la caméra

La capture de la caméra est récupérée par un objet de la classe **DroneCamera**, définie dans le fichier **Scripts/DroneCamera.py**. Cette image est récupérée du rostopic **/iris/camera_red_iris/image_raw** puis convertie en Matrice lisible par la librairie opencv, utilisée par la suite pour détecter un QR code dans l'image puis son centre.

4.3.3 Contrôle du drone

Le drone est contrôlé par un objet de la classe **DroneController**, définie dans le fichier **Scripts/DroneController.py**, ce fichier contient les fonctions permettant d'armer ou désarmer le drone, en changeant la valeur du service **/mavros/cmd/arming**, et de faire décoller et se poser le drone en changeant le mode du pilote automatique via le service **/mavros/set_mode** .

4.3.4 Re-positionnement et rencontre de la plaque et du drone

Après s'être envolé à faible altitude, le drone détecte la position du QR code dans l'image renvoyée par sa caméra.

Si un code est détecté, on détermine si son centre fait partie de la zone de tolérance. Cette zone est un carré dont la taille est donnée par l'utilisateur lors du lancement du script Drone.py avec l'option '-t' ou '–tolerance'.
Si le centre du drone n'est pas inclus dans la zone de tolérance alors on cherche de quel coté de l'image il se trouve puis le drone se déplace dans le sens opposé de la distance définie par l'option '-s' ou '–step'.

Lorsque le centre du code est inclus dans la zone de tolérance le drone augmente son altitude jusqu'à ce que le capteur de distance du drone renvoie une assez faible distance avec la plaque.

Ensuite, l'arrêt et l'attraction du drone vers la plaque sont simulés par la mise en pause de la simulation, solution à laquelle nous avons abouti faute de temps pour créer un plugin Gazebo simulant un électroaimant. Comme cité précédement, l'utilisateur peut alors relancer la simulation en appuyant sur **"u"**.

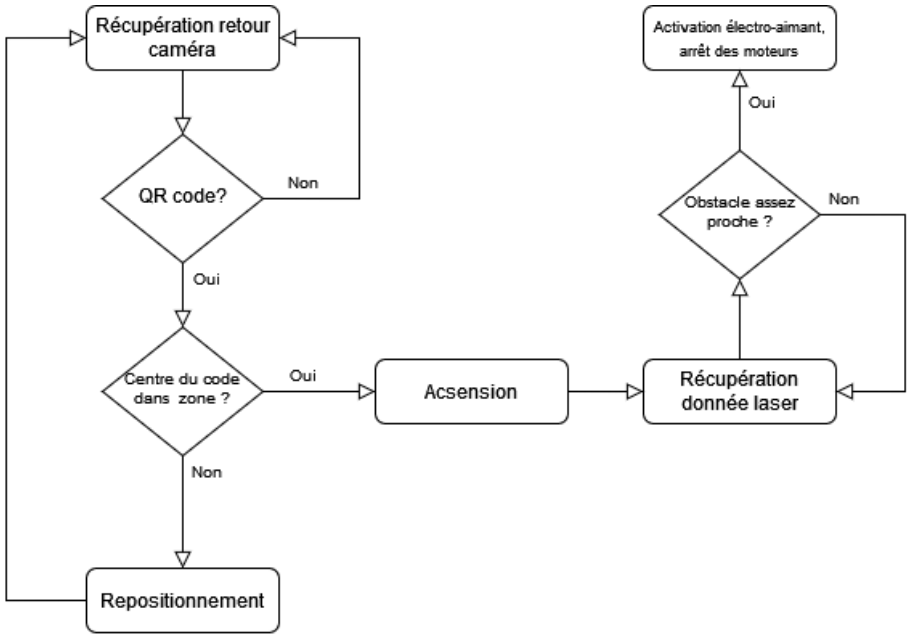


FIGURE 4.1 – Algorithme de repositionnement et du drone et de prise de contact avec la plaque.

4.4 Analyse des tests

Le fichier de tests a été réalisé en utilisant le module unittest de python, pour comprendre son fonctionnement la documentation est disponible via https://docs.python.org/3/library/unittest.html.

Notre objectif en réalisant ces tests a été d'avoir une vérification du bon fonctionnement de tout ce que nous utilisions pour manipuler le drone. De ce fait nos tests peuvent être regroupés dans différentes catégories.

* **Subscribers :**
  . Comme expliqué plus haut le lien entre le code et l'appareil se fait en utilisant un système de topics. Depuis le code on peut soit publier sur un topic pour envoyer des infos au drone soit en recevoir ce sont les Subscribers. Un subscriber transmet les informations correspondant au topic que l'on souhaite, pour tester son bon fonctionnement nous avons alors réalisé une fonction **test_subscribers**. Cette vérification se fait en deux temps, en premier nous initialisons les abonnements avec des méthodes de callback respectives de tous les topics.
  Chaque fois que les topics envois des informations elles arrivent via les fonctions de callback, de ce fait si toutes les méthodes sont appelées on

peut en déduire que les Subscribers sont opérationnels. En sachant ça, notre code utilise un dictionnaire avec une clé et une valeur à False pour chaque topic que l'on veut vérifier, quand une fonction est appelé elle met à jour sa valeur dans le dictionnaire à True. Après un certain temps définit si toutes les valeurs sont à True alors le test réussit sinon il échoue

* **Arming / Disarming :**
  Les tests d'armement et de désarmement du drone sont les plus simple dans leur principe car il s'agit simplement d'envoyer la demande au drone et de vérifier via un service d'état de l'appareil s'il est bien armé ou désarmé.
* **Position :**
  Il est important de s'assurer que quand on transmet des coordonnées au drone, celui-ci se déplace bel et bien à cette position et avec une précision suffisante. Pour réaliser ce test nous envoyons des coordonnées à l'appareil avec un nombre d'itérations suffisantes pour maximiser la précision et une fois fini, nous vérifions via la différence entre la position demandée et la position réelle. Le test n'aboutit que si le résultat est suffisamment petit.
* **Modes :**
  Comme pour les subscribers la fonction de test des changements de modes se base sur une modification d'un dictionnaire si le mode est le bon. Le principe est de parcourir les clés de ce dictionnaire qui correspondent aux modes que l'on veut tester. Et pour chacune des clés envoyer le changement au drone, attendre et vérifier qu'il est bien dans le mode demandé. Si c'est le cas, on peut mettre à jour la valeur du dictionnaire, et à la fin le test est un succès seulement si toutes les valeurs ont été mises à jour.

4.5 Limites et défauts

Le principale défaut de notre simulation concerne l'utilisation d'un électro-aimant, cela est dû au fait que Gazebo ne contient pas de modèle de ce type de composant et qu'avec les délais de notre projet nous n'avons pas le temps d'en créer un de toutes pièces.
Cela dit ce problème n'existe que dans la simulation car l'intégration sur du matériel d'un électro-aimant est assez aisée et est donc un frein uniquement dans la simulation. Une solution alternative pour visualiser le rendu souhaité a quand même était ajoutée.

4.6 Difficultés rencontrées

Lorsque nous avons décidé de réaliser une simulation sur ROS, nous avions sous-estimé la charge de travail liée à la mise en place et la prise en main de ROS et Gazebo. C'est ce qui nous a motivé à développer une image de conteneur Docker afin d'éviter à quiconque souhaite voir le projet d'installer tous les outils nécessaires à son fonctionnement.

Chapitre 5

Partage du code et déploiement du logiciel

5.1 Motivation

Ce projet nécessite une installation de divers outils complexes, le partager s'avère donc complexe. Afin de remédier à cette contrainte nous avons entrepris la conception d'une image de conteneur Docker automatisant l'installation de ROS, Gazebo, et PX4. Le Dockerfile est partagé dans le dossier /**Docker** et l'image est publique et disponible à l'adresse https://hub.docker.com/repository/docker/tdeporte/ros_sitl_gazebo.
L'image est complète et permet d'utiliser Gazebo et l'environnement PX4, ainsi que notre simulation. Toutes les étapes nécessaires à son utilisation sont présentes dans le **README** du projet.

5.2 Analyse du fonctionnement

L'image prend comme source l'image **osrf/ros :noetic-desktop-full** qui prend en charge l'installation d'un environnement **Ubuntu 20.04** sur lequel est installé **ROS noetic**.

Notre Dockerfile prend donc en charge :

* Installation de Gazebo
* Les ressources nécessaires à l'environnement PX4
* Les ressources nécessaires à nos scripts
* Mise en place d'un espace de travail catkin
* Installation de l'environnement PX4
* Clonage et build de nos fichiers

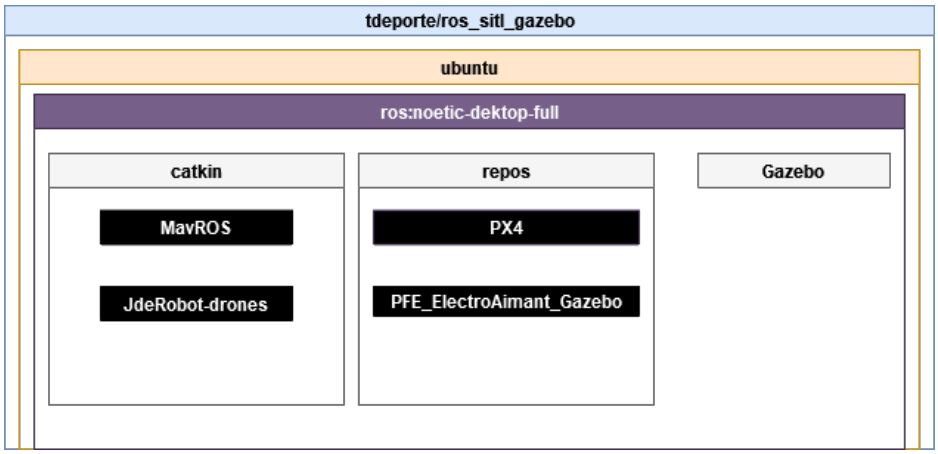


FIGURE 5.1 – Représentation du conteneur

Chapitre 6

Conclusion

Ce projet avait pour ambition de développer un système complet permettant d'offrir une solution à une problématique de déploiement d'un drone en intérieur en optimisant l'espace dans lequel il opère. Le système souhaité étant un drone capable de se fixer et de se détacher d'un support placé au plafond, possible avec l'utilisation d'une caméra et d'un électro-aimant.
Suite à la mise en place d'une simulation, nous avons pu vérifier que l'utilisation de la caméra pour détecter un QR Code placé sur la plaque est fonctionnel ainsi que la fixation grâce à l'électro-aimant. En conclusion, notre projet permet bien d'obtenir un système capable d'offrir un support de déploiement fiable et pratique.
La prochaine étape du projet serait de porter tout le système sur un drone et potentiellement d'améliorer son concept en ajoutant un repérage de la plaque dans une pièce complète.
Pour un usage industriel, plusieurs drones pourraient opérer dans un espace fermé et se partager l'espace dédié à leur déploiement, une grande plaque métallique au plafond, habillée d'un QR code.

Bibliographie

[1] Artifactz. *Laser*.
(Forum, dernière consultation le 16 mars 2022)
https://discuss.px4.io/t/sonar-ir-laser-sensor-in-sitl-gazebo-simulation/6341.

[2] DabitIndustries. *ROS et Opencv*.
(Article, dernière consultation le 3 mars 2022)
https://dabit-industries.github.io/turtlebot2-tutorials/14b-OpenCV2_Python.html.

[3] Nikolas Engelhard. *ROS et unittest*.
(Documentation, dernière consultation le 23 février 2022)
http://wiki.ros.org/unittest.

[4] Jerôme Laplace. *Présentation de Robot Operating System*.
(Article, dernière consultation le 3 février 2022)
https://generationrobots.developpez.com/tutoriels/presentation-robot-operating-system/.

[5] Vanessa Mazzari. *ROS – Robot Operating System*.
(Article, dernière consultation le 3 février 2022)
https://www.generationrobots.com/blog/fr/ros-robot-operating-system-3/.

[6] Python. *Unittest documentation*.
(Documentation, dernière consultation le 23 février 2022)
https://docs.python.org/3/library/unittest.html.

[7] Robotics Simulation Services. *ROS Gazebo : Everything You Need To Know*.
(Article, dernière consultation le 7 février 2022)
https://roboticsimulationservices.com/ros-gazebo-everything-you-need-to-know/.

[8] Wikipédia. *PX4 autopilot*.
(Article, dernière consultation le 11 février 2022)
https://en.wikipedia.org/wiki/PX4_autopilot.